

Unit test/TDD lab for the course: DA616 HT 2015, Malmö Högskola.

Magnus.Krampell@mah.se

1. Introduction

This is a lab aimed at giving an introduction to Unit Tests and TDD (Test-Driven Development).

- Unit testing is a central concept within development and test, where the developer (most often) creates test cases and executes these.
- Unit tests are often created and executed within a test framework (e.g. JUnit)
- **Unit tests with JUnit will be tried in this lab**
- TDD (Test-Driven Development) is a method, where test cases are written before the tested code has been written. The test cases thus fails when executed the first time. The tested code is then written and the test cases are re-run. This time the test cases should pass.
- Test cases written in a test framework can be used as a basis for automatic tests. Whenever a piece of code has been modified, the corresponding tests can be re-run to make sure test cases that passed earlier still pass.
- Automatic tests are useful when performing refactoring – restructuring the code without changing its interface or functionality.
- **TDD will be tried in this lab**

2. Assessment

When you have done both parts of the lab, report for you lab assistant for assessment. The assessment will look at:

- Your answers to the questions (see below)
- Your test cases (code in JUnit) for part 1
- Your test cases (code in JUnit) for part 2
- Your code for implementing the Clock (including Time and Date) in part 2

If you do not finish in time to be assessed during the lab, your results (see list above) shall be uploaded to It's Learning (in the assignment for the lab).

3. Overview

In the first part of the lab, a piece of code is given and the task is to write Unit test cases for it. The JUnit framework, including a tool to measure code coverage, shall be used.

Test cases shall be written so that a certain level of code coverage is reached. The test cases should also reveal any defects in the code. Once these defects have been fixed, the test cases shall pass.

In the second part of the lab, a specification is given. Test cases shall be written and executed. They shall fail when executed. The specification shall then be implemented and test cases re-run. They shall pass when executed.

4. Part 1

a. Task

Start up the development environment (e.g. Eclipse for Java) and get acquainted with the test framework (JUnit) and the code coverage tool used (a special button is used instead of the “Run” button).

QUESTION (3a1): What type of code coverage is measured by the tool (statement, branch, path or condition)? Is it possible to choose more than one type?

b. Task

Load the code rovar.java from It’s learning (unit test lab) into the development environment and make sure it compiles. The code converts strings to and from “rövarspråket” – all consonants are duplicated and an “o” is inserted between them. vowels, numbers and other characters are kept as they are. “rovar” thus becomes “rorovovaror”.

Rövarspråket (English: The Robber Language) is a [Swedish language game](#). It became popular after the books about [Kalle Blomkvist](#) by [Astrid Lindgren](#), where the children use it as a [code](#), both at play and in solving actual crimes.

The principle is easy enough. Every consonant (spelling matters, not pronunciation) is doubled, and an *o* is inserted in-between. Vowels (and other characters, e.g. numbers) are left intact. It is quite possible to render the *Rövarspråket* version of an [English](#) word as well as a [Swedish](#), e.g.: *sos-tot-u-bob-bob-o-ror-non* or *sostotubobboborornon*

that syllable chain would mean *stubborn*. Needless to say, the code is not very useful in written form, but it can be tough when spoken by a trained (and thus quick) user. On the other hand, for an untrained speaker, a word or phrase can often be something of a [tongue-twister](#) or a [shibboleth](#).

Today, the books (and subsequent films) are well known in Sweden, so the language has become integrated in the culture of schoolchildren. Most Scandinavians are familiar with it.

c. Task

Create a test module in JUnit and create a set of test cases (min 3+3) that tests the method(s) of the tested code. Use different test methods to find suitable test data to be used.

Make sure you have considered the following:

- Have you checked all characters? Lower case/upper case? Numbers? Other characters?
- Have you checked empty string? Null pointer?

The following web site may be useful for creating the strings:

<http://utilitymill.com/utility/Rovarspraket>

QUESTION (3c1): What is the maximum code coverage you reach when executing your tests?

QUESTION (3c2): Which defects did you find?

QUESTION (3c3): How did you correct the defects?

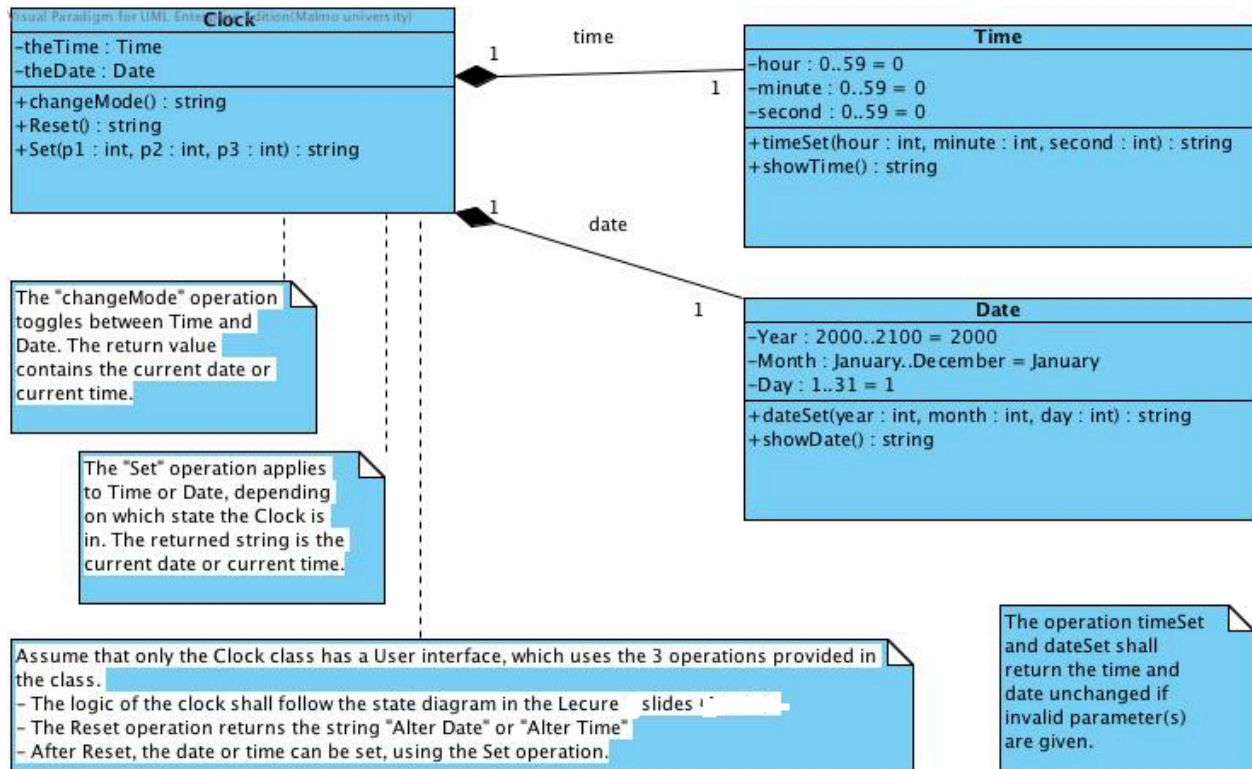
QUESTION (3c4): Do all test cases pass once the defects have been corrected?

5. Part 2

The functionality and corresponding state machine for this task has been presented in the Lecture (including state diagrams). It is assumed that the student is acquainted with the state machine.

a. Task 2.1

Consider the following class diagram:



The task is to test the **3 methods** forming the interface of the Clock class. The Time and Date classes are internal and shall NOT be tested directly – only through the Clock class.

No user interface is needed. The JUnit test shall call the methods and check that the expected result is given in each situation.

Where the specification is not clear, ask or make decisions about how your implementation shall work (e.g. what is returned from Reset? What happens when you try something invalid?)

b. Task 2.2

First create the Clock class, but only containing empty implementations of the 3

methods. (all methods shall return null).

c. Task 2.3

Then create enough test cases in the test framework (JUnit) to test the Clock class.

- Create one test case for **each state transition test** you need (see also question 4c1 below).
- Create **test cases for each border in the Time and Date object**.
- Remember to check for default values in the Time and Date objects.
- Test that the **correct operations can (only) be performed in the correct state** – see table in the lecture slides.

Consider test methods presented in the lectures so far and create test cases so that the clock class is well tested.

Note! The Time and Date classes shall only store the time and date set so that a set time or date can be tested as the returned result. (i.e. they shall **NOT update time and date from the operating system** automatically)

QUESTION (4c1): How many state transitions (compare 0-switch, 1-switch, etc. mentioned in the lecture) are reasonable to test in this case?

d. Task 2.4

Run all tests. All test cases should fail. If they do not, update them!

e. Task 2.5

Implement the functionality of the clock, including the time and date classes and 7 related methods in the classes. Do not overwork, but consider error conditions. Make sure to consider the state!

f. Task 2.6

Run the tests again. All tests should now pass. If they do not, correct the implementation of the clock so that the tests pass. If the test was incorrect, correct it and think about why you created an erroneous test case.

QUESTION (4f1): How much code coverage do you reach with your tests?

g. Task 2.7

Add test cases and Run the tests again until you reach 100% statement coverage or >80% branch coverage.